

Generalizing the Liveness Based Points-to Analysis

Uday P. Khedker and Vini Kanvar*

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Email: {uday,vini}@cse.iitb.ac.in

Abstract

The original liveness based flow and context sensitive points-to analysis (LFCPA) is restricted to scalar pointer variables and scalar pointees on stack and static memory. In this paper, we extend it to support heap memory and pointer expressions involving structures, unions, arrays, and pointer arithmetic. The key idea behind these extensions involves constructing bounded names for locations in terms of compile time constants (names and fixed offsets), and introducing sound approximations when it is not possible to do so. We achieve this by defining a grammar for pointer expressions, suitable memory models and location naming conventions, and some key evaluations of pointer expressions that compute the named locations. These extensions preserve the spirit of the original LFCPA which is evidenced by the fact that although the lattices and the location extractor functions change, the data flow equations remain unchanged.

1 Introduction

The liveness based flow and context sensitive pointer analysis (LFCPA) [4] was proposed as a way of containing the combinatorial explosion of points-to information. Intuitively, it enabled garbage collection over points-to information—the points-to pairs that are guaranteed to be unusable are removed from the points-to relations. This is achieved by restricting the propagation of points-to information to the live ranges of pointers which eliminates the points-to information of pointers that are dead.

For simplicity of exposition, LFCPA was formulated for stack/static memory and scalar pointers although the implementation handled heap, structures, arrays, and pointer arithmetic conservatively but in somewhat adhoc manner. This paper is an attempt to formalize the extensions of LFCPA to heap memory, structures, arrays, pointer arithmetic, and unions. The key idea behind these extensions involves constructing bounded names for locations in terms of compile time constants (names and fixed offsets), and introducing sound approximations when it is not possible to do so. We achieve this by defining a grammar for pointer expressions, suitable memory models and location naming conventions, and some key evaluations of pointer expressions that compute the named locations.

Our naming conventions are based on using variables names, allocation site names, and sequences involving field names or constant offsets. All these are used to create bounded names of the memory locations of interest. Our approximations include using allocation site names, using a collection of named locations, and dropping field names to approximate unions field insensitively. The actual naming conventions and approximations depends on the choice of memory model. The evaluations of pointer expressions include computing their l- and r-values and computing the set of pointers dereferenced to reach the l- or the r-value.

*Vini has been partially supported by the TCS Research Fellowship.

The focus of this paper is on declarative formulations of the extensions rather than efficient algorithms for computing the points-to information. Further, this paper should be seen as a follow up work of LFCPA rather than an independently understandable description. We provide only a brief summary of the original LFCPA in Section 2; please see [4] for more details of the original formulation of LFCPA. Section 3 extends it to support the heap memory and structures (in heap, stack, and static memory) and Section 4 adds the treatment of arrays and pointer arithmetic to the formulation. Section 5 extends our formulation to handle C style unions. Section 6 concludes the paper.

2 Original LFCPA

The definitions presented in this section have been excerpted from the original formulation [4] and have been presented without any explanation.

Given relation $A \subseteq P \times V$ (either Ain_n or $Aout_n$) we first define an auxiliary extractor function

$$Must(A) = \bigcup_{x \in P} \{x\} \times \begin{cases} V & A\{x\} = \emptyset \vee A\{x\} = \{?\} \\ \{y\} & A\{x\} = \{y\} \wedge y \neq ? \\ \emptyset & \text{otherwise} \end{cases} \quad (1)$$

Extractor functions for statement n are $Def_n, Kill_n, Ref_n \subseteq P$ and $Pointee_n \subseteq V$. The data flow values are $Lin_n, Lout_n \subseteq P$ and $Ain_n, Aout_n \subseteq P \times V$.

The extractor functions are defined as follows. We assume that $x, y \in P$ and $a \in V$. A abbreviates Ain_n .

Stmt.	Def_n	$Kill_n$	Ref_n		$Pointee_n$
			if $Def_n \cap Lout_n \neq \emptyset$	Otherwise	
$use\ x$	\emptyset	\emptyset	$\{x\}$	$\{x\}$	\emptyset
$x = \&a$	$\{x\}$	$\{x\}$	\emptyset	\emptyset	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$\{y\}$	\emptyset	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$\{y\} \cup (A\{y\} \cap P)$	\emptyset	$A(A\{y\} \cap P)$
$*x = y$	$A\{x\} \cap P$	$Must(A)\{x\} \cap P$	$\{x, y\}$	$\{x\}$	$A\{y\}$
other	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

The data flow equations are:

$$Lout_n = \begin{cases} \emptyset & n \text{ is } E_p \\ \bigcup_{s \in succ(n)} Lin_s & \text{otherwise} \end{cases} \quad (2)$$

$$Lin_n = (Lout_n - Kill_n) \cup Ref_n \quad (3)$$

$$Ain_n = \begin{cases} Lin_n \times \{?\} & n \text{ is } S_p \\ \left(\bigcup_{p \in pred(n)} Aout_p \right) \Big|_{Lin_n} & \text{otherwise} \end{cases} \quad (4)$$

$$Aout_n = ((Ain_n - (Kill_n \times V)) \cup (Def_n \times Pointee_n)) \Big|_{Lout_n} \quad (5)$$

3 Extending LFCPA to Support Heap Memory and Structures

We describe the proposed extensions by modelling pointer expressions that involve structures and heap, and by defining the lattices and flow functions. The data flow equations remain unchanged, only the flow functions are redefined.

3.1 Modelling Pointer Expressions and the IR

Original LFCPA has been formulated for pointers in stack and static memory. Thus it has a simple model of memory and pointer expressions. The locations are named using variables and pointer expressions have only $*$ and $&$ operators. In order to extend it to support heap data and structures on stack, static area, and heap, we consider three kinds of pointers for a C like language and introduce the sets that we use in our notation.

- A pointer variable $x \in P$ allocated on stack or in static data area.
- A pointer contained in a heap location $o_i \in H$.
- A pointer contained in a field f of a struct variable. The space for this variable may be allocated on stack, static area, or heap.
 - If accessed directly, such a pointer appears as $x.f$ where $x \in (V - P)$, and $f \in pF$. In general, it can be a string like $x.p.q.f$ where $x \in (V - P)$, p and q are non-pointer fields in npF , and $f \in pF$.
 - When accessed through pointers, it appears as $x \rightarrow f$ where $x \in P$, and $f \in pF$. It can be a sequence like $x \rightarrow p \rightarrow q \rightarrow f$ where $x \in P$, and $p, q, f \in pF$.

In general, we can have combinations of the two forms. For Java, however, we have only the latter form and $npF = \emptyset$.

We require the control flow graph form of the IR to be analysed. Although we do not need the statements to be in a 3-address code format, we assume the following two simplifications. In both the cases, theoretically, it may be possible to formulate an analysis that does not need the normalizations. However, it is not desirable for the reasons described below.

- Expressions involving the $&$ operator.

The operand of an *addressof* operator $&$ is required to be an l-value but its result is not an l-value. Hence expressions consisting of $&$ operators cannot be combined orthogonally. Multiple occurrences of $&$ are either illegal or are superfluous due to the presence of $*$ and \rightarrow operators which would be required to make the expression semantically correct. We assume that the expressions involving $&$ are simplified so that there is a single $&$ which occurs in the beginning of an expression.¹

Intuitively, the $&$ and $*/\rightarrow$ operators have “opposite” semantics in terms of the memory graph; the former identifies predecessors of a node in the memory graph while the latter identifies the descendants of a node in the memory graph. Combining both in the analysis is possible but would complicate the formulation.

- Assignment statements $\alpha = \alpha'$ where both α and α' are structures (their types must be same).

¹GCC simplifies pointer expressions containing $&$ in such a manner and we expect all compilers would do the same.

We assume that such an assignment statement has been replaced by the assignments generated by the closure as defined below. The order of these statements is immaterial.

$$closure(\alpha = \alpha') = \begin{cases} \bigcup closure(\alpha.m = \alpha'.m) & \alpha \text{ and } \alpha' \text{ are structures} \\ & \text{and } m \text{ is a field of } \alpha \\ \{\alpha = \alpha'\} & \text{otherwise} \end{cases} \quad (6)$$

Formulating this as a part of analysis changes control flow graph and describing these changes would complicate the formulations.

The other option is to handle this implicitly by describing the effect of the transformations. Our formulation analyses pointer expressions in terms of their parts. This amounts to dividing the expressions in smaller parts. Handling structure assignments requires us to create larger expressions from smaller expressions. While it is possible to combine both reductions and expansions in an analysis, it would complicate the formulation.

The following grammar² defines a pointer expression α which may appear in other expressions, conditions, or assignment statements in a program. These expressions are defined in terms of $x \in V$, $f \in pF \cup npF$, and call to *malloc* function for memory allocation.

$$\alpha := malloc \mid \&\beta \mid \beta \quad (7)$$

$$\beta := x \mid \beta.f \mid \beta \rightarrow f \mid *\beta \quad (8)$$

If $f \in pF$ or $x \in P$, then β is a pointer. Besides, for $\beta \rightarrow f$ and $*\beta$, β must be a pointer. In other cases, β may not be a pointer. When α is *malloc* or $\&\beta$, it does not have an l-value can appear as a pointee only.

3.2 Memory Model and the Lattice of Pointer-Pointee Relations

Since the heap memory is unbounded, we abstract the heap locations based on allocation sites. The memory chunk allocated by an assignment statement $n: x = malloc()$ is named o_n where n is the label of the statement. We define H to contain such names. Note that o_n is a compile time constant and the set H fixed.

We define S_p to contain named locations corresponding to the pointers within structures, and S_m to contain named locations for all members of structures. For brevity, let $R = (V - P) \cup H$ represent the root of a named location.

$$S_p = R \times npF^* \times pF \quad (9)$$

$$S_m = R \times npF^* \times (pF \cup npF) \quad (10)$$

For convenience, we denote a named location consisting of a tuple (a, b, c, d) by concatenating the names as $a.b.c.d$. Some examples of named locations are:

- $a.g.h.f \in S_p$ where $a \in R$ (because $a \in V - P$); $g, h \in npF$; and $f \in pF$.
- $o_1.h.g \in S_m$ where $o_1 \in R$ (because $o_1 \in H$), and $g, h \in npF$.

A field $f \in pF$ cannot appear in the middle because a pointer field cannot have subfields.

It is easy to see that a named locations consist of compile time constants.

Let S denote the set of all pointers and T denote the set of all pointees. Thus S forms the source set and T , the target set, in a points-to relation $A \subseteq S \times T$.

$$S = P \cup H \cup S_p \quad (11)$$

$$T = V \cup H \cup S_m \cup \{?\} \quad (12)$$

Observe that S does not contain pointer expressions but their named locations; similarly, T does not contain expressions describing pointees but their named locations. Figure 1 illustrates the named locations of pointer expressions.

²This grammar is ambiguous; we use the precedences and associativities of the operators as defined for C to disambiguate it.

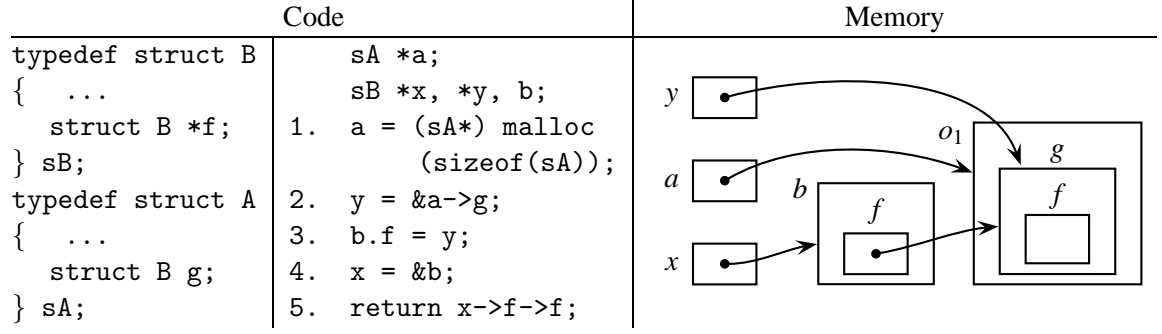


Figure 1: Pointer expressions and their named locations. The named location of x , $x \rightarrow f$, and $x \rightarrow f \rightarrow f$ are x , $b.f$, and $o_1.g.f$ respectively, where o_1 is a heap location derived from its allocation site.

The lattice for data flow variables $Ain_n/Aout_n$ is $(2^{S \times T}, \supseteq)$ whereas the lattice for $Lin_n/Lout_n$ is $(2^S, \supseteq)$. The extractor functions $Def_n, Kill_n, Ref_n$ compute subsets of S .

3.3 Flow Functions

Since our pointer expressions have a rich structure now, we define some auxiliary functions including those that compute the l- and r-values of pointer expressions. These auxiliary functions are then used to define the extractor functions which extract the pointers used in a given statement in the IR.

Computing the l- and r-values of pointer expressions

Given a points-to relation $A \subseteq S \times T$ and an expression α involving pointers, we define the following auxiliary functions that evaluate a pointer expression α in the environment of points-to relation A . These functions are defined recursively using structural induction on the grammar rules thereby ensuring that they cover all possibilities.

We use the following notation: α and β are pointers expressions; field f may be a pointer or non-pointer (i.e. $f \in pF \cup npF$) and σ is either a name in V or H , or a string representation of the name tuple in S_p or S_m .

- $lval(\alpha, A)$ computes the set of possible l-values corresponding to α . Some pointer expressions such as $\&x$ and $malloc$ do not have an l-value.

$$lval(\alpha, A) = \begin{cases} \{\sigma\} & (\alpha \equiv \sigma) \wedge (\sigma \in V) \\ \{\sigma.f \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta.f \\ \{\sigma.f \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv \beta \rightarrow f \\ \{\sigma \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv *\beta \\ \emptyset & \text{otherwise} \end{cases} \quad (13)$$

The pointer expressions $\beta \rightarrow f$ and $*\beta$ involve pointer indirection at the outermost level (i.e. pointees of β are read) and hence we use $rval(\beta)$. This covers the case when β is a pointer expression that does not have an l-value (eg. $(\&x) \rightarrow f$, or $*(\&x)$).

- $rval(\alpha, A)$ computes the set of possible r-values corresponding to α . These are the contents (i.e. the pointees) of the l-values of α . These are defined only if the l-values of α are pointers or α has no

α	$A = \{(a, o_1), (y, o_1.g), (x, b), (b.f, o_1.g), (o_1.g.f, ?)\}$			
	$lval(\alpha, A)$	$rval(\alpha, A)$	$deref(\alpha, A)$	$ref(\alpha, A)$
x	$\{x\}$	$\{b\}$	\emptyset	$\{x\}$
a	$\{a\}$	$\{o_1\}$	\emptyset	$\{a\}$
$*a$	$\{o_1\}$	\emptyset	$\{a\}$	$\{a, o_1\}$
$*x$	$\{b\}$	\emptyset	$\{x\}$	$\{x, b\}$
$*y$	$\{o_1.g\}$	\emptyset	$\{y\}$	$\{y, o_1.g\}$
$b.f$	$\{b.f\}$	$\{o_1.g\}$	\emptyset	$\{b.f\}$
$a \rightarrow g$	$\{o_1.g\}$	\emptyset	$\{a\}$	$\{a, o_1.g\}$
$y \rightarrow f$	$\{o_1.g.f\}$	$\{?\}$	$\{y\}$	$\{y, o_1.g.f\}$
$x \rightarrow f$	$\{b.f\}$	$\{o_1.g\}$	$\{x\}$	$\{x, b.f\}$
$x \rightarrow f \rightarrow f$	$\{o_1.g.f\}$	$\{?\}$	$\{x, b.f\}$	$\{x, b.f, o_1.g.f\}$

Figure 2: Examples of $lval$, $rval$, $deref$, and ref , for some pointer expressions corresponding to Figure 1.

l-value but is $\&\beta$ or $malloc$.

$$rval(\alpha, A) = \begin{cases} lval(\beta, A) & \alpha \equiv \&\beta \\ \{o_i\} & \alpha \equiv malloc \wedge o_i = get_heap_loc() \\ A(lval(\alpha, A) \cap S) & \text{otherwise} \end{cases} \quad (14)$$

Function $get_heap_loc()$ uses the label of the current statement to create a name for heap location allocated by statement. We leave the details implicit for simplicity.

- $deref(\alpha, A)$ computes the set of pointers that need to be read to reach the locations in $lval(\alpha, A)$.

$$deref(\alpha, A) = \begin{cases} deref(\beta, A) & \alpha \equiv \beta.f \\ lval(\beta, A) \cup deref(\beta, A) & (\alpha \equiv \beta \rightarrow f) \vee (\alpha \equiv *\beta) \\ \emptyset & \text{otherwise} \end{cases} \quad (15)$$

Since pointer expressions $\beta \rightarrow f$ and $*\beta$ involve pointer indirection at the outermost level we include the l-values of β also in $deref$.

- $ref(\alpha, A)$ computes the set of pointers which would be read to extract the values in $rval(\alpha, A)$.

$$ref(\alpha, A) = \begin{cases} deref(\beta, A) & \alpha \equiv \&\beta \\ deref(\alpha, A) \cup (lval(\alpha, A) \cap S) & \text{otherwise} \end{cases} \quad (16)$$

Figure 2 illustrates these functions for some pointer expressions corresponding to the program fragment in Figure 1.

For the original LF CPA, the auxiliary functions reduce to the following:

$$lval(\alpha, A) = \begin{cases} \{x\} & (\alpha \equiv x) \wedge (x \in V) \\ \{\sigma \mid \sigma \in rval(x, A), \sigma \neq ?\} & \alpha \equiv *x \\ \emptyset & \text{otherwise} \end{cases} \quad (17)$$

$$rval(\alpha, A) = \begin{cases} lval(x, A) = \{x\} & \alpha \equiv \&x \\ A(lval(\alpha, A) \cap S) & \text{otherwise } (\alpha \equiv *x \text{ or } \alpha \equiv x) \end{cases} \quad (18)$$

$$deref(\alpha, A) = \begin{cases} lval(x, A) & \alpha \equiv *x \\ \emptyset & \text{otherwise} \end{cases} \quad (19)$$

$$ref(\alpha, A) = \begin{cases} deref(x, A) & \alpha \equiv \&x \\ deref(\alpha, A) \cup (lval(\alpha, A) \cap S) & \text{otherwise } (\alpha \equiv *x \text{ or } \alpha \equiv x) \end{cases} \quad (20)$$

Extractor functions used in the data flow equations

An abstract heap object may represent multiple concrete heap objects thus prohibiting strong updates. The predicate $heap(l)$ asserts that a location l is on heap.

$$heap(l) \Leftrightarrow l \in H \cup H \times npF^* \times (pF \cup npF) \quad (21)$$

Given a points-to relation $A \subseteq S \times T$, we define its must version using this predicate as follows:

$$Must(A) = \bigcup_{p \in S} \{p\} \times \begin{cases} T & (A\{p\} = \emptyset \vee A\{p\} = \{?\}) \wedge \neg heap(p) \\ \{q\} & A\{p\} = \{q\} \wedge q \neq ? \wedge \neg heap(q) \wedge \neg heap(p) \\ \emptyset & \text{otherwise} \end{cases} \quad (22)$$

Let A denote Ain_n and mA denote $Must(A)$. Then, the extractor functions Def_n , $Kill_n$, Ref_n , and $Pointee_n$ are defined for various statements as follows.

- Pointer assignment statement $lhs_n = rhs_n$. We assume that this statement is type correct and both lhs_n and rhs_n are pointers.

$$Def_n = lval(lhs_n, A) \quad (23)$$

$$Kill_n = lval(lhs_n, mA) \quad (24)$$

$$Ref_n = \begin{cases} deref(lhs_n, A) & Def_n \cap Lout_n = \emptyset \\ deref(lhs_n, A) \cup ref(rhs_n, A) & \text{otherwise} \end{cases} \quad (25)$$

$$Pointee_n = rval(rhs_n, A) \quad (26)$$

Observe the use of mA (i.e. $Must(A)$) in the definition of $Kill_n$.

- *Use* α statement. This statement models all uses of pointers which are not in a pointer assignment statement (eg. $x \rightarrow n = 10$; where n is an integer field of a structure).

$$Def_n = Kill_n = Pointee_n = \emptyset \quad (27)$$

$$Ref_n = ref(\alpha, A) \quad (28)$$

- Any other statement.

$$Def_n = Kill_n = Ref_n = Pointee_n = \emptyset \quad (29)$$

Stmt. n	Def_n	$Kill_n$	Ref_n	$Pointee_n$
1	$\{a\}$	$\{a\}$	\emptyset	$\{o_1\}$
2	$\{y\}$	$\{y\}$	$\{a\}$	$\{o_1.g\}$
3	$\{b.f\}$	$\{b.f\}$	\emptyset	$\{o_1.g\}$
4	$\{x\}$	$\{x\}$	\emptyset	$\{b\}$
5	\emptyset	\emptyset	$\{x, b.f, o_1.g.f\}$	\emptyset

Figure 3: Extractor Functions for the statements in Figure 1. These have been computed using the final liveness and points-to information presented in Figure 4.

Stmt. n	Lin_n	$Lout_n$	Ain_n	$Aout_n$
1	$\{o_1.g.f\}$	$\{a, o_1.g.f\}$	$\{(o_1.g.f, ?)\}$	$\{(a, o_1), (o_1.g.f, ?)\}$
2	$\{a, o_1.g.f\}$	$\{y, o_1.g.f\}$	$\{(a, o_1), (o_1.g.f, ?)\}$	$\{(y, o_1.g), (o_1.g.f, ?)\}$
3	$\{y, o_1.g.f\}$	$\{b.f, o_1.g.f\}$	$\{(y, o_1.g), (o_1.g.f, ?)\}$	$\{(b.f, o_1.g), (o_1.g.f, ?)\}$
4	$\{b.f, o_1.g.f\}$	$\{x, b.f, o_1.g.f\}$	$\{b.f, o_1.g, (o_1.g.f, ?)\}$	$\{(x, b), (b.f, o_1.g), (o_1.g.f, ?)\}$
5	$\{x, b.f, o_1.g.f\}$	\emptyset	$\{(x, b), (b.f, o_1.g), (o_1.g.f, ?)\}$	\emptyset

Figure 4: Final round of liveness and points-to analysis for the statements in Figure 1.

The data flow equations (2)–(5) remain unchanged.

Observe that if we exclude heap and structures and restrict α to scalar pointers in P , definitions (23)–(29) of Def_n , $Kill_n$, Ref_n , and $Pointee_n$ reduce to the definitions in the original LFCPA formulations.

Figure 3 shows the values of the extractor functions for the final round of analysis (i.e., they are computed using the final liveness and points-to information).

4 Handling Pointer Arithmetic and Arrays

So far our pointers in aggregates are restricted to structures. Now we extend them to include arrays as well as pointer arithmetic. Given an arithmetic expression $e \in E$, the grammar for extended pointer expressions is as follows. For simplicity we have considered only $+$ operator for pointer arithmetic; $-$ operator can be handled by negating the value of e .

$$\alpha := \text{malloc} \mid \&\beta \mid \beta \mid \&\beta + e \quad (30)$$

$$\beta := x \mid \beta.f \mid \beta \rightarrow f \mid *\beta \mid \beta[e] \mid \beta + e \quad (31)$$

In general, $e \in E$ may be defined in terms of variables whose values may not be known at compile time. Let $\llbracket e \rrbracket \in C$ represent the evaluation of expression e where C is a set of constants. If e cannot be evaluated at compile time, then $\llbracket e \rrbracket = \perp_{\square}$ which is included as a fictitious constant in C .

4.1 Memory Model and the Lattice of Pointer-Pointee Relations

The inclusion of array accesses and pointer arithmetic goes against the grain of our memory model which, so far, is not addressable. In other words, we access a location by a name and not by an address in the current model—functions $lval$ and $rval$ return a name rather than an address. Such a model is based on an implicit assumption that two different names refer to two different locations. The operators of pointer expressions

<pre> struct s1 { int i; int *h; }; struct s2 { int y; struct s1 g[10]; }; </pre>	<pre> struct s3 { int x; struct s2 f; }; struct s3 a[20][10]; </pre>
---	--

Figure 5: An example of complex nestings of arrays and structures. We model the location of pointer $a[7][3].f.g[5].h$ by the name $a.7.3.f.g.5.h$.

preserve this invariant. However, when we combine names and pointer arithmetic or array indexing, the operators of pointers expressions cannot preserve this invariant. As a consequence, our model may not be able to find out, and hence cannot guarantee, that the actual locations of pointers $x + 4$ and y are different. Such checks would require an addressable model of memory where each name (including the names created using *lval*) is at an offset from a start address; the offset for the location of a pointer expression can be computed by reducing the pointer arithmetic and using the offsets of fields and array elements.

While this may work well for static and stack locations, it would not work for heap because we would not know the offsets of dynamic memory allocation at compile time. Hence we discard the addressable model³ and use a partially addressable model in which only the locations within an array are addressable with respect to the name of the array and no names overlap in memory even if they use an offset. For ensuring soundness of may points-to analysis, we use the following approximations:

- Whenever an offset for an array location cannot be computed at compile time, we view all accesses to the array as index-insensitive and treat the entire aggregate as a single variable. For soundness, read and write accesses would need different approximations based on this assumption.
 - A read would be approximated as reading *any* location.
 - A write would be approximated as writing into *any* location for the purpose for generating liveness or points-to information and writing into *no* location for the purpose of killing liveness or points-to information.
- When pointer arithmetic is used, we assume that the resulting location could coincide with
 - *any* location within the array if the pointer points to an array, or
 - *any* of the named locations if the pointer does not hold the address of an array.

With this model, we extend the structure pointers S_p and the structure member S_m (equations (9) and (10)) to define general pointers and members G_p and G_m . Since we can have structures within arrays and arrays within structures, we allowing field names to be interspersed with constant offsets. Let C represent the set of constant offsets. Then,

$$G_p = R \times (C \cup npF)^* \times (C \cup pF) \quad (32)$$

$$G_m = R \times (C \cup npF)^* \times (C \cup pF \cup npF) \quad (33)$$

Figure 5 illustrates our modelling. Given two names $x.c_1.c_2$ and $y.f.g$, the computation of actual memory locations for them would be different: y is a structure and a compiler would simply add the offsets of f and g to the address of y to get the actual location of $y.f.g$. However, in the name $x.c_1.c_2$, x is an array (because it

³It may be possible to use separate memory models for stack/static memory and heap.

is followed by a number c_1) and the array address calculation performed by the compiler is not just addition of c_1 and c_2 . Instead, c_1 will have to be multiplied by the number of columns and then c_2 will be added.

The offsets appearing in the named locations are unscaled; we do not use a scaling factor based on the size of the data type. Our goal is to uniquely identify locations for pointer-pointee relations rather than for accessing the memory. For the latter, the offsets will have to be scaled up using data type. Since we assume a typed IR, this information should be available in the preceding field/variable name in the list.

Observe that the named locations continue to be compile time constants.

The definitions of source and target sets remain same as equations (11) and (12) except that S_p and S_m are replaced by G_p and G_m respectively.

$$S = P \cup H \cup G_p \quad (34)$$

$$T = V \cup H \cup G_m \cup \{?\} \quad (35)$$

4.2 Extractor Functions

The revised definition of $lval$ appears below. When compared with equation (13), it is clear that the only change is handling an array access; the result of pointer arithmetic does not have an l-value. The l-value of a pointer expression $\beta[e]$, is defined by appending the evaluation of e to the l-value of β . In case of multi-dimensional arrays, the l-value of β would already have a suffix containing some offsets. The new case in the definition of $lval$ has been marked in blue.

$$lval(\alpha, A) = \begin{cases} \{\sigma\} & (\alpha \equiv \sigma) \wedge (\sigma \in V) \\ \{\sigma.f \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta.f \\ \{\sigma.f \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv \beta \rightarrow f \\ \{\sigma \mid \sigma \in rval(\beta, A), \sigma \neq ?\} & \alpha \equiv * \beta \\ \{\sigma.[e] \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta[e] \\ \emptyset & \text{otherwise} \end{cases} \quad (36)$$

We allow \perp_{\square} to appear in a name string. In such a situation, it is interpreted as *any* location in that dimension of the array (as described in Section 4.1) except for *Must* and *Kill* where it is interpreted as *no* location as defined in equations (40), and (41) below.

The default case of $rval$ covers the array accesses and we only need to add rules for handling pointer arithmetic. The most common use of pointer arithmetic is to access array elements through pointers. In such cases, pointer increments are well defined and pointer arithmetic has a predictable behaviour. In other cases of pointer arithmetic, it is difficult to find out the exact r-values of a pointer expression.

Consider a pointer expression $x + c$. If x points to an array location, the name of its r-value would have constant offset as its suffix (because the r-value of x is an array location). In order to discover the r-value of $x + c$, we simply need to add the scaled value of c to the offset with a scaling factor governed by the size of the type of values held by the array. This is easily generalized to $\beta + e$ in the definition of $rval$. In all other cases of $\beta + e$, we approximate the r-values by *any* location denoted by the universal set of pointees T . Similarly, the r-values of $\&\beta + e$ are also approximated by the universal set of pointees T . The new cases have been marked in blue.

$$rval(\alpha, A) = \begin{cases} lval(\beta, A) & \alpha \equiv \&\beta \\ \{o_i\} & \alpha \equiv malloc \wedge o_i = get_heap_loc() \\ T & (\alpha \equiv \beta + e) \wedge (\exists \sigma \in rval(\beta, A), \sigma \neq \sigma'.c, \sigma' \in T, c \in C) \\ \bigcup \{\sigma.(c + [e])\} & (\alpha \equiv \beta + e) \wedge (\sigma.c \in rval(\beta, A)) \wedge (c \in C) \\ A(lval(\alpha, A) \cap S) & \text{otherwise} \end{cases} \quad (37)$$

<pre>char *** t, ** s; char * q[10], p; t = &s; s = &q[3]; q[8] = &p;</pre>	<p>Pointer s points to an array. The points-to information after these statements is $\{(t, s), (s, q.3), (q.8, p)\}$.</p> <ul style="list-style-type: none"> • The <i>lval</i> of pointer expressions $*(t + 5)$, $*(s + 5)$, $*(\&q[3] + 5)$, and $q[8]$ after these statements is $\{q.8\}$. • The <i>rval</i> of the same expressions is $\{p\}$.
<pre>int *** a, ** b; int * c, * d; a = &b; b = &c;</pre>	<p>The points-to information after these statements is $\{(a, b), (b, c)\}$.</p> <ul style="list-style-type: none"> • The <i>lval</i> of pointer expressions $*(a + 5)$, $*(b + 5)$, and $*(\&c + 5)$ is $T - \{?\}$. • The <i>rval</i> of the same expressions is T.

Figure 6: Examples of evaluations of pointer expressions involving arrays and pointer arithmetic.

Since the default value of *deref* is \emptyset , we need to add the cases $\beta[e]$ and $\beta + e$ explicitly.

$$deref(\alpha, A) = \begin{cases} deref(\beta, A) & (\alpha \equiv \beta.n) \vee (\alpha \equiv \beta[e]) \vee (\alpha \equiv \beta + e) \\ lval(\beta, A) \cup deref(\beta, A) & (\alpha \equiv \beta \rightarrow m) \vee (\alpha \equiv * \beta) \\ \emptyset & \text{otherwise} \end{cases} \quad (38)$$

The pointers read to reach the pointees of $\beta + e$ are included in the definition of *rval* as shown below.

$$ref(\alpha, A) = \begin{cases} deref(\beta, A) & (\alpha \equiv \&\beta) \vee (\alpha \equiv \&\beta + e) \\ ref(\beta, A) & \alpha \equiv \beta + e \\ deref(\alpha, A) \cup (lval(\alpha, A) \cap S) & \text{otherwise} \end{cases} \quad (39)$$

The definition of *Kill* should exclude the points-to relations of approximated arrays. We achieve this by defining a by a predicate *approx*(l) which asserts that l is a heap location or involves \perp_{\square} . The revised definitions of *Kill* and *Must* are:

$$Must(A) = \bigcup_{p \in S} \{p\} \times \begin{cases} T & (A\{p\} = \emptyset \vee A\{p\} = \{?\}) \wedge \neg \text{approx}(p) \\ \{q\} & A\{p\} = \{q\} \wedge q \neq ? \wedge \neg \text{approx}(q) \wedge \neg \text{approx}(p) \\ \emptyset & \text{otherwise} \end{cases} \quad (40)$$

$$Kill_n = \{\sigma \mid \sigma \in lval(lhs_n, mA), \neg \text{approx}(\sigma)\} \quad (41)$$

All other definitions remain same except that \perp_{\square} is interpreted as *any* location.

Figure 6 illustrate compile time names for pointer expressions involving arrays and pointer arithmetic.

5 Handling Unions

With support for handling structures, handling C style unions becomes straight forward. We conservatively assume that when $\beta.f$ refers to access of field f in a union, it could coincide with

- *any* field name for the purpose of generating the liveness and points-to information, and with
- *no* field name for the purpose of killing the information.

```

union {
  struct {
    int * g;
  } f[30];
  int * h[20];
} a, b, * c[10];

c[4] = &a;
c[5] = &b;

```

- The *lval* of the following pointer expressions evaluates to a:
a.f[0].g, a.h[1], c[4]->f[2].g, and c[4]->h[3].
- The *lval* of the following pointer expressions evaluates to b:
b.f[6].g, b.h[7], c[5]->f[8].g, and c[5]->h[9].

Figure 7: An example of complex nestings of arrays, structures, and unions to illustrate accesses of union fields and their compile time names created using field-insensitive approximation.

The first requirement is served by approximating a union field-insensitively—we drop the field names appearing in a union. The second requirement is served by ensuring that $\text{union}(\sigma) \Rightarrow \text{approx}(\sigma)$ for equations (40) and (41).

The revised definition of *lval* uses a predicate $\text{union}(\sigma)$ which asserts that σ is a union. The changes have been marked in blue.

$$\text{lval}(\alpha, A) = \begin{cases} \{\sigma\} & (\alpha \equiv \sigma) \wedge (\sigma \in V) \\ \begin{cases} \{\sigma.f \mid \sigma \in \text{lval}(\beta, A), \neg \text{union}(\sigma)\} \\ \cup \{\sigma \mid \sigma \in \text{lval}(\beta, A), \text{union}(\sigma)\} \end{cases} & \alpha \equiv \beta.f \\ \begin{cases} \{\sigma.f \mid \sigma \in \text{rval}(\beta, A), \sigma \neq ?, \neg \text{union}(\sigma)\} \\ \cup \{\sigma \mid \sigma \in \text{rval}(\beta, A), \sigma \neq ?, \text{union}(\sigma)\} \end{cases} & \alpha \equiv \beta \rightarrow f \\ \{\sigma \mid \sigma \in \text{rval}(\beta, A), \sigma \neq ?\} & \alpha \equiv * \beta \\ \begin{cases} \{\sigma.[e] \mid \sigma \in \text{lval}(\beta, A), \neg \text{union}(\sigma)\} \\ \cup \{\sigma \mid \sigma \in \text{lval}(\beta, A), \text{union}(\sigma)\} \end{cases} & \alpha \equiv \beta[e] \\ 0 & \text{otherwise} \end{cases} \quad (42)$$

Figure 7 illustrates approximate names for unions.

6 Conclusions

With a suitable choice of naming conventions in terms of compile time constants, and a suitable choice of functions to compute their l- and r-values, extending LFCPA to support heap memory, structures, arrays, and pointer arithmetic seems a relatively a straight forward extension.

We have chosen to retain the spirit of declarative formulation of the original LFCPA and have not addressed the issue of efficient algorithms for the formulations. We have described our extensions in the intraprocedural setting. It remains to be seen whether it is feasible to extend them to interprocedural level using the default method of value contexts [1, 3, 5] as was done in the original LFCPA or whether some additional issues need to be addressed. Further, implementation of this method and empirical measurements would be a non-trivial exercise and is left as future work.

We have handled heap memory using the allocation site based abstraction in this paper. It would be interesting to see how the use-site based abstraction as defined in HRA [2] can be used in our extensions. Besides, the possibility of using an addressable model for stack and static memory with some other model for heap can also be explored.

Acknowledgments

We thank Swati Jaiswal and Pritam Gharat for their feedback in the early stages of the formulations in this paper.

References

- [1] U. P. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Proceedings of the International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2008.
- [2] U. P. Khedker, A. Sanyal, and A. Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems*, 30(1):1, 2007.
- [3] U. P. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group), 2009. (Under publication).
- [4] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In *Proceedings of the 19th International Conference on Static Analysis, SAS’12*, pages 265–282, Berlin, Heidelberg, 2012. Springer-Verlag.
- [5] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis, SOAP ’13*, pages 31–36, New York, NY, USA, 2013. ACM.